Informix®

Informix 4GL and Informix SQL

A Guide to the Version 7.20 and 7.30 Feature Enhancements

Abstract

This Informix® 4GL and Informix SQL Feature Enhancements document describes the new features within the version 7.x releases of these tools. It reviews the enhancements and gives instruction for using them. Note that when a reference to Informix 4GL is made, that includes both the compiled Informix 4GL c-code compiler and the Rapid Development System p-code compiler.

Table of Contents

Introduction

Informix 4GL 7.30 and Informix SQL 7.30 are the latest versions of the Informix family of character-based tools in the UNIX environment. This release is an upgrade to the 6.x family of English and Asian Informix 4GL and Informix SQL products. A recompile of the source code is necessary to use this new compiler.

These products have all of the functional enhancements and big fixes of the 6.x family of Informix 4GL and Informix SQL products. In addition, these products support the following functional enhancements:

Beginning with version 7.20:

- Stream Pipe mode of communication between the client and Informix Dynamic Server™ engine.
- Support for new environment variables:
  - DBCENTURY
  - FET_BUF_SIZE
  - PDQPRIORITY
- Support for the 7.20 SQL syntax in 4GL and SQL through the PREPARE statement.
- GLS environment variables such as:
  - GL_DATE
  - GL_DATETIME

Beginning with version 7.30:

- Global string space—Releases prior to version 7.30 supported a total of no more than 64 KB in all of the names of variables in a single 4GL program. In version 7.30, that limit is now approximately 2 GB. This allows larger applications to be constructed.
- I4GL compiler changes that result in a smaller executable.
- DBCENTURY fields—Before version 7.30, DBCENTURY was set at the environment level for an entire program. This allows DBCENTURY settings at the field-level within a single application.
- New 4GL language extensions include:
  - SQL grammar extension—extending supported SQL syntax to version 9.2+ through use of an SQL...END SQL block.
  - Direct support for CONNECT, SET CONNECTION, and DISCONNECT statements.
  - Support for the concatenation string operator—provides functionality of the standard “double-pipe” string concatenation operator ("||").

- Control INSERT and DELETE operations in INPUT ARRAY—allows the INSERT and DELETE keys to be enabled or disabled independent of each other.
- New ATTRIBUTE formats in INPUT ARRAY—support for two new functions to set attribute formats inside a dialog box.
- Dynamically configure size of report—allows the size of the report page and the report destination to be specified when a report starts.
- Dynamic control of effective size of a program array (INPUT ARRAY)—allows the size of the INPUT ARRAY to be dynamically controlled.
- Current row highlighted automatically—allows the current row to be highlighted without requiring any code changes.
- Get size of screen array—declares the size of a named screen array so that the correct number of values is displayed.
- COMMENT OFF in windows—allows a window with a form to be reduced to one or two (an input and comment line) lines.

Enhancements in connectivity-stream pipe

In previous versions of these products, when an Informix server and the Informix 4GL and Informix SQL products were installed on the same computer, the tools products and the server were limited to the following modes of communication:

- Shared memory
- Network (local loopback)

The version 7.20 Informix 4GL and Informix SQL adds the stream-pipe interprocess communications (IPC) mechanism.

Unlike shared-memory connections, stream-pipe connections do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.

It should also be noted that stream-pipe connections have the following disadvantages:

- Slower than shared memory connections
- Not available on all computers

For stream-pipe connections, the entry in the nettype field of the sqlhosts file is onipcstr for an Informix server connection:

```
--------------------------------------
server nettype    hostname  service
--------------------------------------
alpha    onipcstr  idcsun33  service1
--------------------------------------
```

Support for additional environment variables

This section describes the new environment variables supported in this release and provides examples for each variable.

DBCENTURY

Until this release, Informix 4GL and Informix SQL products used the current century to extend the year in a DATE or DATETIME value with less than three digits for the **year** field.

With **DBCENTURY** support, it is possible to choose the present, previous, or next century to extend the year. New algorithms have been introduced for this purpose.

To indicate the algorithm to be used for extending the year, set the **DBCENTURY** environment variable to one of the letters listed in the following table. Note that the variable is case sensitive and if it is unset or mis-set, then the default is used. Also, the current century means that in 1999, 1900 is added; and from 2000 to 2099, 2000 is added.

P	Past
F	Future
C	Closest
R	Current (default)

There are two ways to specify the algorithm for the century extension:

- Explicitly set the **DBCENTURY** environment variable
- Use the default **DBCENTURY** value, which is *R* (current)

DBCENTURY=P (Past)

With this setting, the past and present centuries are used to extend the year value. These two dates are compared with the current date, and the date prior to the current date is chosen. If both dates are prior to the current date, the one that is closest is chosen.

Date Type	Current Date	User Enters	Previous Century	Present Century	Result
Example 1					
DATE	4/6/1996	1/1/10	1/1/1810	1/1/1910	1/1/1910
Analysis: Both results are prior to the current date, but 1/1/1910 is closer.					
Example 2					
DATE	4/6/2010	1/1/05	1/1/1905	1/1/2005	1/1/2005
Analysis: Both dates are prior to the current date, but 1/1/2005 is closer.					
Example 3					
DATE	4/6/2010	1/1/50	1/1/1950	1/1/2050	1/1/1950
Analysis: Only 1/1/1950 is prior to the current date.					
Example 4					
DATETIME YEAR TO MONTH	4/6/1996	DATETIME(1-1) YEAR TO MONTH	1801-1	1901-1	1901-1
Analysis: Both dates are prior to the current date, but 1901-1 is closer.					

DBCENTURY=F (Future)

With this setting, the present and next centuries are used to extend the year value. These two dates are compared with the current date, and then the date after the current date is chosen. If both dates are after the current date, the one that is closest is chosen.

Date Type	Current Date	User Enters	Previous Century	Present Century	Result
Example 1					
DATE	4/6/1996	1/1/90	1/1/1990	1/1/2090	1/1/2090
Analysis: 1/1/2090 is after the current date.					
Example 2					
DATE	4/6/2051	1/1/50	1/1/2050	1/1/2150	1/1/2150
Analysis: 1/1/2150 is after the current date.					
Example 3					
DATE	4/6/2010	1/1/25	1/1/2025	1/1/2125	1/1/2025
Analysis: Both dates are after the current date, but 1/1/2025 is closer.					
Example 4					
DATETIME YEAR TO MONTH	4/6/1996	DATETIME(1-1) YEAR TO MONTH	1901-1	2001-1	2001-1
Analysis: 2001-1 is after the current date.					

DBCENTURY=C (Closest)

With this setting, the past, present, and next centuries are used to extend the year value, and the date closest to the current date is used.

Data Type	Current Date	User Enters	Previous Century	Present Century	Result
Example 1					
DATE	4/6/1996	1/1/1	1/1/1801 to 4/6/1996 is 71,318 days	1/1/1901 to 4/6/1996 is 34,794 days	1/1/2001
Analysis: 1/1/2001 is closer to the current date.					
Example 2					
DATE	2/1/2001	4/1/95	4/1/1995 to 2/1/2001 is 2,133 days	4/1/2095 to 2/1/2001 is 34,392 days	4/1/1995
Analysis: 4/1/1995 is closer to the current date.					

Example 3

Data type:	DATETIME YEAR TO MONTH
Current date:	4/6/1996
User enters:	DATETIME(1-1) YEAR TO MONTH

The number of months from the current date is calculated:

Previous century extension:	1801-1 to 1996-4 is 2,343 months
Present century extension:	1901-1 to 1996-4 is 1,143 months
Next century extension:	2001-1 to 1996-4 is 57 months
Result:	2001-1
Analysis:	2001-1 is closer to the current date

DBCENTURY=R (Current)

With this setting, the current century is used to extend the year.

Data Type	Current Date	User Enters	Result
Example 1			
DATE	4/6/1996	12/10/1	12/10/1901
Example 2			
DATE	4/6/2010	12/10/1	12/10/2001
Example 3			
DATETIME YEAR TO MONTH	4/6/1996	DATETIME(1-12) YEAR TO MONTH	1901-12

FET_BUF_SIZE
The *fetch buffer* is the buffer that the server uses to send data (except for blob data) to the client applications. The buffer resides in the client process. The bigger the buffer, the more data the server can send to the application before returning control to the application. For a large number of rows to be returned, or for a few large rows to be returned, the greater the size of the fetch buffer, the fewer number of times the application needs to wait while the server retrieves the rows. This generally results in improved performance because the overhead of refilling the client-side buffer is reduced.

You can set the size of the fetch buffer by setting the **FET_BUF_SIZE** environment variable to the desired value anytime prior to runtime. Setting this environment variable sets the fetch buffer size for the duration of the application.

For example, the following command sets the variable in a C-shell environment to 20,000 bytes (20 KB):

```
setenv FET_BUF_SIZE 20000
```

The size should be in bytes, up to a maximum value for a small integer (short) of the system used. For most 32-bit systems, the value is 32,767 bytes. If the parameter is not set externally, a default value is used.

For more details, see the *Informix ESQL/C Programmer's Manual* version 9.21, page 14-22.

PDQPRIORITY

The PDQPRIORITY value determines the amount of resources an Informix Dynamic Server allocates to process a query in parallel.

The following are examples of appropriate syntax in the C-shell environment

setenv PDQPRIORITY HIGH/LOW/OFF/percent-of-resources

The options to this command include:

HIGH	The server determines respective degree of parallelism
LOW	(Default) Data is fetched from partitioned tables in parallel
OFF	Parallel data query is turned off
Percentage-of-resources	An integer between 0 and 100, indicating the percentage of available parallel resources that should be allocated for this particular program or query

An example of the different ways to explicitly set PDQPRIORITY inside a 4GL program includes:

```
DEFINE setpdq CHAR(255)

-- Directly embedded SQL (4.10 Syntax only)
DATABASE stores7

-- The SET PDQPRIORITY statement was added in OnLine 6.00
-- It cannot be written as a directly embedded SQL statement
LET setpdq = "set pdqpriority low"

-- Prepared and Executed Statement (all versions of I4GL)
PREPARE p_setpdq FROM setpdq
EXECUTE p_setpdq
FREE p_setpdq

-- Alternative way of executing statement in a string
-- Only available from version 7.30
EXECUTE IMMEDIATE setdeb

-- Any preparable SQL statement can be written in an SQL
-- block (enclosed in SQL … END SQL)
SQL
    SET PDQPRIORITY LOW
END SQL
```

Using 7.2-level SQL syntax in 4GL

Informix 4GL 7.20 directly supports the SQL syntax of legacy Informix 4.1 database servers. In addition, the PREPARE statement can be used to support all Informix server 7.2-level SQL statements, with the *exception* of the statements listed in the following tables:

Data Definition Statements	**Connection**
CREATE PROCEDURE FROM	CONNECT
	DISCONNECT
Data Manipulation Statements	SET CONNECTION
LOAD	
UNLOAD	**Data Integrity**
	CHECK TABLE
Cursor Manipulation	REPAIR TABLE
CLOSE	
DECLARE	**Auxiliary Statements**
FETCH	INFO
FLUSH	GET DIAGNOSTICS
FREE	OUTPUT
OPEN	WHENEVER
PUT	SET DEFERRED PREPARE
Dynamic Management	
ALLOCATE DESCRIPTOR	
DEALLOCATE DESCRIPTOR	
DESCRIBE	
EXECUTE	
EXECUTE IMMEDIATE	
FREE	
GET DESCRIPTOR	
PREPARE	
SET DESCRIPTOR	

GLS environment variables

Informix 4GL and Informix SQL products now provide full support of GLS environment variables, including:

- **GL_DATE**
- **GL_DATETIME**

For details and syntax of the GLS environment variables, see the *Guide to GLS Functionality*. The **glfiles** utility (described in Appendix A of the *Guide to GLS Functionality* and packaged with Informix 4GL and Informix SQL products) allows you to generate a list of:

- GLS locales available in the system
- Informix code-set conversion files available
- Informix code-set files available

Default values of GLS environment parameters

Default values assumed by Informix 4GL and Informix SQL products (which differ from those of ALS environments) are described in this section. Default Values of CLIENT_LOCALE and DB_LOCALE.

The following table shows the values assumed by Informix 4GL and Informix SQL when you define only some of the required values of locales. (A value of ja-jp.ujis is assumed in the following example. CL means CLIENT_LOCALE, and DL means DB_LOCALE.)

User Defined				**Values in Product**	
CL Defined	**CL Value**	**DL Defined**	**DL Value**	**CL Value**	**DL Value**
No	--	No	--	en_us.8859	en_us.8859
Yes	ja_jp.ujis	No	--	ja_jp.ujis	ja_jp.ujis
Yes	ja_jp.ujis	Yes	ja_jp.ujis	ja_jp.ujis	ja_jp.ujis
No	--	Yes	ja_jp.ujis	en_us.8859	ja_jp.ujis

If you do not set the DBLANG environment variable, it is set to the value of CLIENT_LOCALE.

Compatibility of Informix 4GL and Informix-ESQL/C

Because Informix 4GL 7.20 and Informix SQL 7.20 are built with version 7.2x of ESQL/C (which is also part of the 7.2x server products), you can now link the object modules of 7.20 4GL and 7.2x ESQL/C to create binaries.

This is also true for Informix-4GL 7.30, which is built with Informix ClientSDK™ 2.30 (aka ESQL/C 9.21). Developers will be able to employ 9.x functionality in their 4GL application with the use of the new SQL…END SQL code block.

Global string space

Releases of 4GL prior to version 7.30 supported a total of no more than 65,535 bytes in all the names of variables in a single 4GL program, including record members and redefined variables. In this release, however, the upper limit on global string space (which includes variables, regardless of their scope, and certain other named 4GL program entities) is now 2 GB (2,048 MB). This allows you to write larger applications.

Your available system resources might impose a lower limit. In programs that are compiled to p-code, however, a single 4GL function or report can have a total of no more than 32,767 bytes in the names of all its variables.

I4GL compiler changes

The c4gl script, which is used to compile 4GL programs, has been modified. Previously, whenever a 4GL program was compiled, c4gl would link the libraries statically, meaning that all the required libraries were packed into the executable. This release does away with that approach. 7.30 c4gl links these libraries dynamically. *This change dramatically reduces the size of the executable created by c4gl.* If you still require statically linked libraries, use the "-static" option.

Symbols are resolved at runtime, so the operating system must be able to locate the shared library at runtime. This normally means that the environment variable LD_LIBRARY_PATH must be set to include $INFORMIXDIR/lib/tools and $INFORMIXDIR/lib. The name LD_LIBRARY_PATH is for Solaris, Linux, and other platforms, but some platforms use different names. For example, LIBPATH is used by IBM AIX and SHLIB_PATH is used by HP-UX.

DBCENTURY fields

The CENTURY field attribute determines how to expand abbreviated (two-digit, or one digit) *year* values in DATE or DATETIME fields of 4GL forms. These features can also expand a two-digit (or one digit) *year* in a PROMPT statement.

As in version 7.20 of the 4GL and SQL tools, the use of the DBCENTURY environment variable allows DATE and DATETIME fields that have a two-digit year entered to be converted to four digits before further processing. This creates some limitations if different centuries need to be assumed for different fields on the same form. For example, a date-of-birth field is normally historical (babies seldom arrive when predicted), so it would be useful to set DBCENTURY to ‘P’ for such fields. By contrast, a planned retirement date or mortgage pay-off date is normally still in the future, so it would be useful to set DBCENTURY to ‘F’ for such fields. One single value for DBCENTURY cannot be set to both values. The solution in the version 7.30 tools is the use of the CENTURY field attribute.

The CENTURY attribute supports the same settings as the DBCENTURY environment variable and can specify any of the four algorithms to expand abbreviated years into four-digit year values. However, its scope is restricted to a single field. If the CENTURY and DBCENTURY settings are different, CENTURY takes precedence.

An example of this syntax is noted below:

```
f008 = customer.ship_date, CENTURY='C';
```

The ATTRIBUTES clause that can follow the FOR clause of the PROMPT statement can also specify CENTURY as an attribute.

```
PROMPT " Enter date: " for dateval ATTRIBUTE (CENTURY='C')
```

SQL grammar extension

Most SQL statements that Informix 4.10 databases support can be directly embedded in 4GL source code. So can a few of the non-preparable ESLQ/C statements introduced after version 4.10, such as CONNECT, CREATE PROCEDURE FROM, DISCONNECT, FOREACH…WITH REOPTIMIZATION, OPEN…WITH REOPTIMIZATION, and SET CONNECTION. Other SQL statements that include syntax later than version 4.10 must be prepared, if the database server can prepare and execute them.

The SQL…END SQL delimiters provide an alternative facility by which an SQL statement is automatically prepared, executed, and freed. For example, this ALTER TABLE statement includes the DISABLED keyword, which was introduced to the Informix implementation of SQL after the version 4.10 release:

```
SQL
ALTER TABLE cust_fax MODIFY (lname CHAR(15)
NOT NULL CONSTRAINT lname_notblank DISABLED)
END SQL
```

A statement like this, which has no input nor output parameters, is simply placed between the SQL and END SQL keywords. It resembles an embedded SQL statement, except that its post-4.10 syntax would have produced a compilation error if the SQL…END SQL delimiters were absent.

Other examples of how SQL blocks are supported include:

- Parameter-less SQL statements

```
SQL
   CREATE TRIGGER t1_sometable UPDATE
      FOR SomeTable
      REFERENCING NEW AS newrow
      ON EVERY ROW (EXECUTE PROCEDURE
         LogUpdate(“SomeTable”,
            newrow.pk_column))
END SQL
```

- In using ‘$’ to identify 4GL variables for input

```
SQL
   SELECT TRIM(SomeColumn) || ‘-new’
      FROM SomeTable
      WHERE PK_Column = $rec.pk_column
END SQL
```

- In using ‘$’ to identify 4GL variables for output

```
SQL
   EXECUTE PROCEDURE someproc(12, 19)
      INTO $var1, $rec.part2,
         $arr[i].part3
END SQL
```

- In using ‘$’ to identify 4GL variables for both input and output

```
SQL
   EXECUTE PROCEDURE someproc($v1, $v2)
      INTO $var1, $rec.part2,
         $arr[i].part3
END SQL
```

- In using '$' to identify 4GL variables within a cursor

```
DECLARE c_select SCROLL CURSOR
   WITH HOLD FOR SQL
   SELECT TRIM(BOTH SomeCol) || '-old'
      INTO $str
      FROM SomeTable
      WHERE OtherCol > $val
END SQL
```

Direct support for CONNECT, SET CONNECTION, and DISCONNECT

You can now simultaneously connect to different databases. At compile time, you can still only connect to a single database in any given module using the non-procedural database statement. At runtime, you can connect to multiple databases; but, since I4GL does not support multithreading, you can only have one active database at any one time. The others will be dormant. However, it means that your application could be written to extract data from a SE database into program variables and then insert the data into an IDS database, without having to repeatly connect to the two databases. You would simply switch between two connections. Here is a skeletal example of how to connect to two different databases and switch between the two connections before disconnecting:

```
  CONNECT TO 'stores7@dbserver'
       AS $conn1
       USER $username USING $password
       WITH CONCURRENT TRANSACTIONS
  BEGIN WORK
  CONNECT TO "newdbs" AS 'conn2'
WHILE …
  SET CONNECTION $conn1
  …
  SET CONNECTION 'conn2'
  …
END WHILE
SET CONNECTION $conn1
  COMMIT WORK
  DISCONNECT ALL
```

Databases can be specified with a variable and you should always use a VARCHAR to define the password variable.

Support for the concatenation string operator

A new concatenation operator, symbolized by two pipe symbols, "||," has been introduced, which follows the ANSI rules for concatenation.

If any of the operands of "||" is a NULL, then the resultant is a NULL. The "||" operator has a lower precedence than +,-, and a higher precedence than MATCHES. The "||" operator takes precedence over the existing comma concatenation operator used in assignment statements. An example:

```
DEFINE a CHAR (28)
LET a = “Informix “ || “Development “ || “Center”
DISPLAY “a=”,a,”*”
```

This would result in [a=Informix Development Center *]

Control INSERT and DELETE operations in INPUT ARRAY

Controlling INSERT and DELETE key operations on screen arrays:

The new INPUT ARRAY attribute can be set as

```
INSERT ROW            -- signifies that INSERT KEY is enabled
INSERT ROW = TRUE     -- signifies that INSERT KEY is enabled
INSERT ROW = FALSE    -- signifies that INSERT KEY is disabled
```

When INSERT ROW = FALSE is specified, the user cannot use the Insert key to perform insert actions within the INPUT ARRAY statement. (The user can still perform insert actions by using the TAB, ARROW, and RETURN keys in the last initialized row.)

When INSERT ROW = TRUE is specified, the user is not prevented from using the Insert key to enter data. The default is TRUE, which corresponds to the behavior of previous 4GL releases.

```
DELETE ROW            -- signifies that DELETE KEY is enabled
DELETE ROW = TRUE     -- signifies that DELETE KEY is enabled
DELETE ROW = FALSE    -- signifies that DELETE KEY is disabled
```

When DELETE ROW = FALSE is specified, the user cannot perform any DELETE actions within the INPUT ARRAY statement.

When DELETE ROW = TRUE is specified, the user is not prevented from using the Delete key to delete data. The default is TRUE, which corresponds to the behavior of previous 4GL releases.

The following example disables the insert and delete keys on rows of the screen array:

```
INPUT ARRAY arrayname WITHOUT DEFAULTS FROM s_array.*
ATTRIBUTE(INSERT ROW = FALSE, DELETE ROW = FALSE)
```

New ATTRIBUTE formats in INPUT ARRAY

Within an INPUT ARRAY, it is now possible to override the INSERT and DELETE operations in the BEFORE INSERT and BEFORE DELETE clauses respectively:

```
CANCEL INSERT in the 'BEFORE INSERT' clause of INPUT ARRAY
CANCEL DELETE in the 'BEFORE DELETE' clause of INPUT ARRAY
```

As an example, the programmer might want to implement a system where the user is allowed to delete all but one of the rows, but once a row is deleted, a replacement row cannot be inserted in its place. The following code implements this design:

```
DEFINE n_rows INTEGER
DEFINE arrayname ARRAY[100] OF RECORD

     . . .
     INPUT ARRAY arrayname WITHOUT DEFAULTS FROM s_array.*
       ATTRIBUTES(COUNT = n_rows, MAXCOUNT = n_rows,
         INSERT ROW = FALSE, DELETE ROW = TRUE
     BEFORE INSERT
       CANCEL INSERT
     BEFORE DELETE
       LET n_rows = n_rows - 1
       IF n_rows <= 0 THEN
         CANCEL DELETE
       END IF
     END INPUT
```

Dynamically configure size of reports

The dimensions and destination of output from a 4GL report can be specified at runtime, overriding the report definition. Originally, a reports output type and specifications were hardcoded within the source code, which meant that it was not usually possible to have a single report that formatted sensibly both on a screen and on a printer.

Example of how this works:

```
START REPORT reportname
  TO OUTPUT "pipe" DESTINATION "lp -dcypress"
    WITH PAGE LENGTH = 64, TOP MARGIN = 0,
    BOTTOM MARGIN = 0, LEFT MARGIN = 0
```

Typically, the dimensions and destination are all specified through variables rather than constants.

Dynamic control of effective size of a program array (INPUT ARRAY)

In 4GL, all program arrays are static in size. Prior to version 7.30, and INPUT ARRAY statement always allowed the user to insert as many rows as were defined in the array. Now, with version 7.30, the programmer can control the number of the rows that should be used by an INPUT ARRAY using the new MAXCOUNT attribute (and the companion COUNT attribute which supersedes the SET COUNT() function):

```
INPUT ARRAY prog_array WITHOUT DEFAULTS FROM scr_array.*
  ATTRIBUTE( MAXCOUNT = x, COUNT = y)
```

The COUNT attribute can specify the number of records within a program array that contain data when the INPUT ARRAY starts. The MAXCOUNT attribute can specify the dynamic or effective size of the program array. This size can be less than the defined size of the program array, but must not be smaller than the COUNT value.

If MAXCOUNT is specified as less than one or greater than the declared program array size, the original program array size is used as the MAXCOUNT value.

You can specify both COUNT and MAXCOUNT in the same ATTRIBUTE clause:

```
CALL SET_COUNT(5)
INPUT ARRAY prog_array WITHOUT DEFAULTS
FROM scr_array.* ATTRIBUTE( MAXCOUNT = 10, COUNT = 6)
```

In this example, the COUNT attribute overrides the SET_COUNT( ) value. The number of rows initially displayed will be six, but the user will only be able to add up to four new rows, even if the array `prog_array` is defined with 100 elements.

Current row highlighted automatically

A new attribute, CURRENT ROW DISPLAY, has been implemented to highlight the current row during input and output operations of screen arrays, for INPUT ARRAY and DISPLAY ARRAY statements.

Get size of screen array

The function FGL_SCR_SIZE( ) accepts as its argument the name of a screen array in the currently opened form and returns an integer that corresponds to the number of screen records in that screen array.

The built-in FGL_SCR_SIZE( ) function returns the declared size of a specified screen array at runtime. In the following example, a form specification file (called file.per) declares two screen arrays, called s_rec1 and s_rec2:

```
DATABASE FORMONLY
SCREEN
{
[f1 ] [f2 ]
[f1 ] [f2 ]
[f1 ] [f2 ]
[f3 ] [f4 ]
[f3 ] [f4 ]
[f5 ]
}
ATTRIBUTES
f1 = FORMONLY.a ;
f2 = FORMONLY.b ;
f3 = FORMONLY.c ;
f4 = FORMONLY.d ;
f5 = FORMONLY.e ;
INSTRUCTIONS
DELIMITERS “”
SCREEN RECORD s_rec1[3] (a,b)
SCREEN RECORD s_rec2 (c,d)
```

The following 4GL program invokes the FGL_SCR_SIZE( ) function:

```
MAIN
DEFINE n1,n2 INT
   DEFINE ch CHAR(10)
   OPEN WINDOW w1 AT 2,3 WITH FORM “file” ATTRIBUTE BORDER)
   CALL fgl_scr_size(“s_rec1”) RETURNING n1
   LET n1 = fgl_scr_size(“s_rec1”) -- Can also be called
                                   -- in a LET statement
   DISPLAY “n1 = “, n1
   LET ch = “s_rec2”
   CALL fgl_scr_size(ch) RETURNING n2
   LET n2 = fgl_scr_size(ch) -- Can also be called
                             -- in a LET statement
DISPLAY “n2 = “, n2
CLOSE WINDOW w1
END MAIN
```

This program produces the following output:

```
n1 = 3
n2 = 2
```

The proper value is returned even though the array dimension is not specified in the form file.

Obtaining user keyboard entry information

The function FGL_GETKEY( ) waits for a key to be pressed and returns the integer code of the physical key that the user pressed.

The FGL_GETKEY( ) function generates the same codes for keys that the FGL_KEYVAL( ) and FGL_LASTKEY() functions return. FGL_GETKEY( ) can be invoked whenever the user is required to enter a single key stroke that will not be interpreted through the current MENU or INPUT (or CONSTRUCT or DISPLAY ARRAY) statement.

Here is an example of a program fragment that calls both functions, so that FGL_KEYVAL( ) evaluates what FGL_GETKEY( ) returns.

```
DEFINE key INT
PROMPT “Press the RETURN key to continue. “ ||
  “Press any other key to quit.”
LET key = FGL_GETKEY( )
IF key = FGL_KEYVAL(“return”) THEN
  CALL continue()
ELSE
  CALL quit()
END IF
```

COMMENT OFF in windows

When space is tight and a new window needs to be opened in pre-7.30 4GL, the window always had a few extra lines that could not be removed. Starting with version 7.30, the comment line can be turned off in a new window globally throughout the ‘OPTIONS’ section of the program or at an individual window. The result is a smaller window footprint on the screen.

Syntax:

```
COMMENT LINE OFF
```

Example:

```
OPEN WINDOW ..... ATTRIBUTE (MENU LINE 1, PROMPT LINE 1,
       FORM LINE 1, COMMENT LINE OFF, MESSAGE LINE 1)
```

This turns off the comment line in the selected window. This change does not apply to the OPTIONS statement.

Summary

The release of version 7.30 of Informix 4GL and Informix SQL offers significant enhancements to the core language. You can review the latest 4GL and SQL manuals online at www.informix.com/answers. Informix Dynamic 4GL version 3.0 supports all of the enhancements mentioned in this document except for the increased size of the GSS (which was never a problem in earlier versions of Dynamic 4GL). Of course, it also provides the ability to run an application in GUI mode from a Windows PC or through a Web browser.

About Informix

Based in Menlo Park, California, Informix Corporation specializes in advanced information management technologies that help enterprises in the i.Economy get to market quickly, generate new revenue, build a unique strategic advantage, and solve their most complex business problems. Informix offers customers a complete software infrastructure for the Web that delivers highly scalable transaction processing, personalized content management, integrated business intelligence, full multimedia capabilities, and complete e-commerce solutions.

For more information, contact the sales office nearest you or visit our Web site at www.informix.com.

Informix®

4100 Bohannon Drive
Menlo Park, CA 94025
Tel. 650.926.6300
www.informix.com

INFORMIX REGIONAL SALES OFFICES

Asia Pacific	65 298 1716	Japan	81 3 5562 4500
Canada (Toronto)	416 730 9009	Latin America	305 591 9592
Europe/Middle East/Africa	44 208 818 1000	North America	800 331 1763
Federal	703 847 2900		650 926 6300

Printed in U.S.A. 1/00
000-21998-70